

+ 22

Understanding Allocator Impact on Runtime Performance

PARSA AMINI



20
22



Motivation

- “C++ Allocators for the Working Programmer”
 - Book in progress
 - by Joshua Berne and John Lakos
 - Detail allocators performance benefits
 - Teach how to write programs using `std::pmr`
 - Explain the benefits of using local allocators
 - ... much more
 - For anyone writing C++ software

Motivation

- “C++ Allocators for the Working Programmer” – Research
 - Ongoing work
 - Provide empirical evidence
 - Quantify the benefits of using `std::pmr`
 - Justify with quantified impact on hardware
 - The strategies in this talk will be applied to more involved case studies throughout the book

What you will NOT see in this talk

- Allocator design
- Embedded devices, AARCH, PPC, GPUs, etc.
 - We have Intel x86 machines (Gulftown, Sandy Bridge EN, Cascade Lake).
- Alternate memory types (e.g., fast, shared, protected, pinned, I/O mapped)
 - We use plain old anonymous memory.
- Concurrency
 - Currently our code is all single-threaded.
- Instrumented allocators
 - We are not focused on assisting with debugging, profiling, measurement, or testing.
- Alternate global allocators
 - We use GNU allocator (glibc) on Linux.

What you will see in this talk

- Local memory allocator performance analysis
 - Performance measurement
 - Including impact of diffused allocations
 - Performance metrics
 - Execution time and hardware performance counters
- Simulating diffused allocations.
- Existing tools
 - Linux perf
 - Intel VTune Profiler
 - Perfmon2 libpfm4

Allocator Impact on Runtime Performance

- ☐ Allocators and how they can improve performance?
- ☐ Allocator Performance Impact Analysis
- ☐ Results
- ☐ Conclusion

Background: Allocators in C++ programs

- General-purpose global allocators
 - Interface
 - new/delete (malloc/free)
 - Implementation
 - e.g., GNU allocator, TCMalloc, jemalloc, mimalloc, hoard.
- Special-purpose, e.g., local (arena) allocators
 - namespace `std::pmr`
 - Run-time pluggable allocation strategies
 - Special-purpose memory resources
 - e.g., `std::pmr::monotonic_buffer_resource`, `std::pmr::unsynchronized_pool_resource`, and `std::pmr::synchronized_pool_resource`

How allocators can improve performance

- Faster allocations and deallocation
- Better locality

How allocators can improve performance

- Faster allocation and deallocations
 - Fewer calls to new/delete
 - Simpler allocation algorithm
 - Monotonic incrementing
 - Delay freeing until the end of algorithm
 - No-op deallocation
- Better locality

How allocators can improve performance

- Faster allocations and deallocation
- Better locality
 - Better spatial locality
 - Access fewer pages (4KiB or 2MiB)
 - Access fewer cache lines (64 bytes)
 - Invalidate fewer cache lines
 - Fewer page faults
 - e.g., less diffusion
 - Better temporal locality
 - Invalidate fewer cache lines (again)
 - Fewer page faults (again)

What is diffusion?

```
std::vector<std::string> strcol;  
strcol.reserve(4);  
for (std::size_t i = 0; i < 4; ++i) {  
    strcol.emplace_back("some large  
string");  
}  
  
for (std::size_t i = 0; i < 4; ++i) {  
    std::cout  
        << (void*)&(strcol[i].data()[0])  
        << '\n';  
}
```

Compact allocation example:



0xcebf40

0xcebf60 // &strcol[1][0] - &strcol[0][0] = 0x20

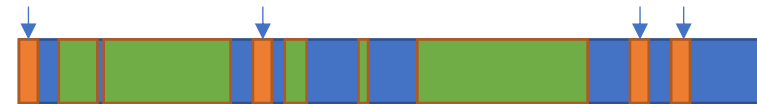
0xcebf80 // &strcol[2][0] - &strcol[1][0] = 0x20

0xcebfa0 // &strcol[3][0] - &strcol[2][0] = 0x20

vs.

Diffused allocations example:

strcol[0] strcol[1] strcol[2] strcol[3]



0x185aeb0

0x513ef0

0x5d2eb0

0xd35ef0

Better locality e.g., Heap fragmentation

- A long-running program may cause it
 - Global heap becomes fragmented
 - new might give diffused memory
- A better choice of general-purpose global allocator helps
 - And is orthogonal to our work
- Local allocators protect against diffused allocations
 - Unlike the global allocator, knows how memory will be used
 - Allocation for a small task come from a small number of large blocks

How allocators can improve performance

- Faster allocations
 - Not always important
 - For programs that allocate and dellocate too often
 - Easy to measure
 - Number of calls visible in any profiler
- Better locality
 - Always important
 - Harder to measure

Allocator Impact on Runtime Performance

- ☒ Allocators and how they can improve performance?
- ☐ Allocator Performance Impact Analysis
- ☐ Results
- ☐ Conclusion

Allocator Performance Impact Analysis

- Performance metrics
 - Execution time and derived metrics e.g., speedup
 - OS – Page faults
 - Hardware performance counters
 - Cache statistics
- Other, software-characterization metrics
 - DVLUC – John Lakos
 - P0089r1 (wg21.link/p0089r1)
 - Meeting C++ 2017 Local (Arena) Memory Allocators (2 parts talk)
 - <https://youtu.be/ko6uyw0C8r0>
 - <https://youtu.be/fN7nVzbRiEk>
- Measurement!
- Our goal
 - Exhibit clearly each relevant phenomenon in isolation.

Allocator Performance Metrics

- Execution time
- Pages
 - Page faults
- Memory reads and stores
 - Cache line accesses
 - Invalidate fewer cache lines

Experiment Requirements

- Allocator implementations
- Programs that showcases allocator usage
- A benchmarking framework to measure those programs performance
- Interface with hardware performance counter
- Machine(s)

Our machines

- Milieu
 - Cascade Lake (2019)
 - L1: 2x 640 KiB (1.2 MiB), L2: 2x 10 MiB (20 MiB), L3: 2x 13.8 MiB (27.5 MiB)
 - Main memory: 64 GiB
 - [Intel Xeon Silver 4210 Processor/13.75M Cache, 2.20 GHz](#)
- Bugg
 - Gulftown (Westmere-EP) (2011)
 - L1: 192 KiB, L2: 1.5 MiB, L3: 12 MiB
 - Main memory: 24 GiB
 - [Intel Core i7-980 Processor/12M Cache, 3.33 GHz, 4.8 GT/s Intel QPI](#)
- Rostam Marvin
 - Sandy Bridge-EN (2012)
 - L1: 2x 512 KiB (1024 KiB), L2: 2x 2 MiB (4 MiB), L3: 2x 20 MiB (40 MiB)
 - Main memory: 48 GiB
 - [Intel Xeon Processor E5-2450/20M Cache, 2.10 GHz, 8.00 GT/s Intel QPI](#)
 - Special thanks to: Hartmut Kaiser at Center for Computation and Technology at Louisiana State University

Benchmarking Framework

- Option 1: Use an existing framework
 - (e.g., Google Benchmark, nanobench, Nonius)
 - Con: One more dependency
- Option 2: Make one
 - AWPBM
 - BDE-style
 - Run subject code in a loop
 - Avoid interfering compiler optimizations
 - Can read hardware performance counters (via libpfm4)

Hardware Performance Counters

- Sample or count supported hardware events on the CPU
 - e.g., Retired cycles, retired instructions, cache statistics, etc.
 - Not every event you want is supported.
- Accessing performance counter requires privilege.
 - x86 CPU privilege level 0.
 - Access involves system calls.

Access Performance Counters

- Portable access interfaces
 - Operating system
 - Linux perf_events API
 - Linux perf tool
 - libpfm4 (perfmon2)
 - Uses perf_events underneath
 - PAPI
 - Uses libpfm4
 - likwid-perfctr (LIKWID)
- Intel VTune Profiler

Intel VTune Profiler

- We deal with only Intel hardware.
 - We can use Intel VTune Profiler.
 - Free under community support license for commercial projects.
- Benefits:
 - Made & maintained by the vendor
 - Helps understanding existing performance counters
 - Higher-level, composite metrics
 - e.g., Memory bound, L1 bound
 - Command-line and web interfaces

VTune - Memory Access Analysis

Elapsed Time[?]: 10.630s

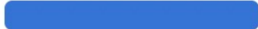
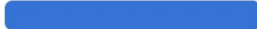
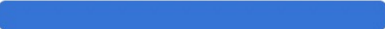

































CPU Time [?] :	10.505s	
▼ Memory Bound[?]:	2.5%	of Pipeline Slots
L1 Bound [?] :	15.2%	of Clockticks
L2 Bound [?] :	0.0%	of Clockticks
L3 Bound [?] :	0.0%	of Clockticks
➤ DRAM Bound[?]:	0.0%	of Clockticks
Store Bound [?] :	0.0%	of Clockticks
NUMA: % of Remote Accesses [?] :	0.0%	
UPI Utilization Bound [?] :	0.0%	of Elapsed Time
Loads:	10,469,803,740	
Stores:	5,084,428,140	
▼ LLC Miss Count[?]:	0	
Local DRAM Access Count [?] :	0	
Remote DRAM Access Count [?] :	0	
Remote Cache Access Count [?] :	0	
Average Latency (cycles) [?] :	7	
Total Thread Count:	3	
Paused Time [?] :	0s	

High-level metrics

Hardware Events				
Hardware Event Type	Hardware Event Count	Hardware Event Sample Count	Events Per Sample	Precise
CPU_CLK_UNHALTED.ONE_THREAD_ACTIVE	253,429,440	16	500015	FALSE
CPU_CLK_UNHALTED.REF_TSC	23,056,000,000	2,096	11000000	FALSE
CPU_CLK_UNHALTED.REF_XCLK	253,429,440	16	500015	FALSE
CPU_CLK_UNHALTED.THREAD	31,361,000,000	2,851	11000000	FALSE
CPU_CLK_UNHALTED.THREAD_P	31,360,999,923	99	10000015	FALSE
CYCLE_ACTIVITY.CYCLES_L1D_MISS	0	0	10000015	FALSE
CYCLE_ACTIVITY.CYCLES_MEM_ANY	23,441,555,498	74	10000015	FALSE
CYCLE_ACTIVITY.STALLS_L1D_MISS	0	0	10000015	FALSE
CYCLE_ACTIVITY.STALLS_L2_MISS	0	0	10000015	FALSE
CYCLE_ACTIVITY.STALLS_L3_MISS	0	0	10000015	FALSE
CYCLE_ACTIVITY.STALLS_MEM_ANY	4,751,666,655	15	10000015	FALSE
CYCLE_ACTIVITY.STALLS_TOTAL	10,453,666,641	33	10000015	FALSE
DTLB_LOAD_MISSES.STLB_HIT:cmask=1	0	0	10000015	FALSE
DTLB_LOAD_MISSES.WALK_ACTIVE	839,485,020	53	500015	FALSE
DTLB_STORE_MISSES.STLB_HIT:cmask=1	0	0	500015	FALSE
DTLB_STORE_MISSES.WALK_ACTIVE	31,678,680	2	500015	FALSE
EXE_ACTIVITY.1_PORTS_UTIL	5,701,999,986	18	10000015	FALSE
EXE_ACTIVITY.2_PORTS_UTIL	5,701,999,986	18	10000015	FALSE
EXE_ACTIVITY.BOUND_ON_STORES	0	0	10000015	FALSE
FRONTEND_RETIRED.LATENCY_GE_4_PS	364,319,379	23	500035	TRUE
IDQ_UOPS_NOT_DELIVERED.CORE	46,883,110,996	148	10000015	FALSE
INST_RETIRED.ANY	38,291,000,000	3,481	11000000	FALSE
INT_MISC.RECOVERY_CYCLES	3,484,555,547	11	10000015	FALSE

List of measured performance counters

Memory Access Analysis: Top-down Tree

Function Stack	MEM_INST_RETIRED.ALL_LOADS_PS	MEM_INST_RETIRED.ALL_STORES_PS	MEM_TRANS_RETIRED.LOAD_LATENCY_GT_4
Total	10,469,803,740 	5,084,428,140 	142,592,490 
▶ BloombergLP::awpcs1::Unique2::uniqu	4,276,621,800 	1,932,399,480 	55,452,635 
▶ std::_Rb_tree_insert_and_rebalance	2,074,953,540 	1,694,809,380 	30,102,859 
▶ std::_Rb_tree<unsigned char, unsigned	1,219,629,180 	443,501,520 	23,765,415 
▶ func@0x186e94	1,187,950,500 	158,393,400 	12,674,888 
▼ func@0x186214	158,393,400 	0	4,753,083 
▼ std::_Rb_tree_decrement	237,590,100 	0	4,753,083 
▼ std::uniform_int_distribution<unsigned	174,232,740 	79,196,700 	3,168,722 
▶ func@0x404320	0	0	1,584,361 
▶ std::pmr::monotonic_buffer_resource::	31,678,680 	126,714,720 	1,584,361 
▶ __libc_malloc	174,232,740 	63,357,360 	1,584,361 
▶ func@0x810f0	158,393,400 	63,357,360 	1,584,361 
▶ func@0x82560	95,036,040 	316,786,800 	1,584,361 
▶ __do_softirq	0	0	0
▶ std::pmr::get_default_resource	0	0	0
▶ func@0xa23a0	15,839,340 	0	0

Intel VTune Profiler

- Enables:
 - Sanity check
 - Understand the application
 - Survey available measurement features
 - Check for the current microarchitecture's available metrics
- Cons:
 - Uses sampling
 - Hard to use directly when applying to a variety of scenarios
- Provides:
 - Number of loads and stores per instruction
 - Memory bound %, LLC miss count, average latency per instruction
- Relevant VTune analysis mode: Memory access

Hardware Performance Counter Annoyances

- Some counters work in sampling mode only.
 - Use for profiling
- Precision varies across counters.
- Understanding what the counter measures.
- Linux kernel panic mode must be set appropriately to access counters.
- Not every counter you want is available on hardware you have.
- Need to handle performance counter overflows.
- Not every counter is known by the performance counter access interface.
 - Use raw events ids.
 - Use a wrapper like ocperf.

Hardware Performance Counter Annoyances

- Not every counter you want is available on hardware you have.
 - Reminder: Clock speeds have not changed since the aughts (2000s).
 - For our purposes results are portable across machines.
 - Get results from a different Intel architecture that has that counter.
- Backup solution in the case of caches: Use a simulator
 - Cachegrind (valgrind)

Allocator Implementations

- `std::pmr` allows choose allocation algorithm at runtime for the same code
- Baseline: `std::pmr::new_delete_resource`
- Monotonic Allocator: `std::pmr::monotonic_buffer_resource`

Case Study 1

- The program count unique characters
 - Arguments: One `std::string_view`
 - Return: A `std::size_t` with the number of unique characters
- Variable: Number of characters in the passed `std::string_view`
- Contenders:
 - Use a `std::set`
 - Use a `std::pmr::set, monotonic_buffer_resource`
 - Use a `std::pmr::set, monotonic_buffer_resource` with a local buffer
 - With or without a monotonic allocator

1.1 - Baseline: Use std::set

```
std::size_t countUniqueChars1(const std::string_view& s)
{

    std::set<char> uniq;
    uniq.insert(s.begin(), s.end());
    return uniq.size();
}
```

1.2 - Contender : Use std::pmr::set

```
std::size_t countUniqueChars2(const std::string_view& s)
{
    std::pmr::monotonic_buffer_resource mr;

    std::pmr::set<char> uniq(&mr);
    uniq.insert(s.begin(), s.end());
    return uniq.size();
}
```

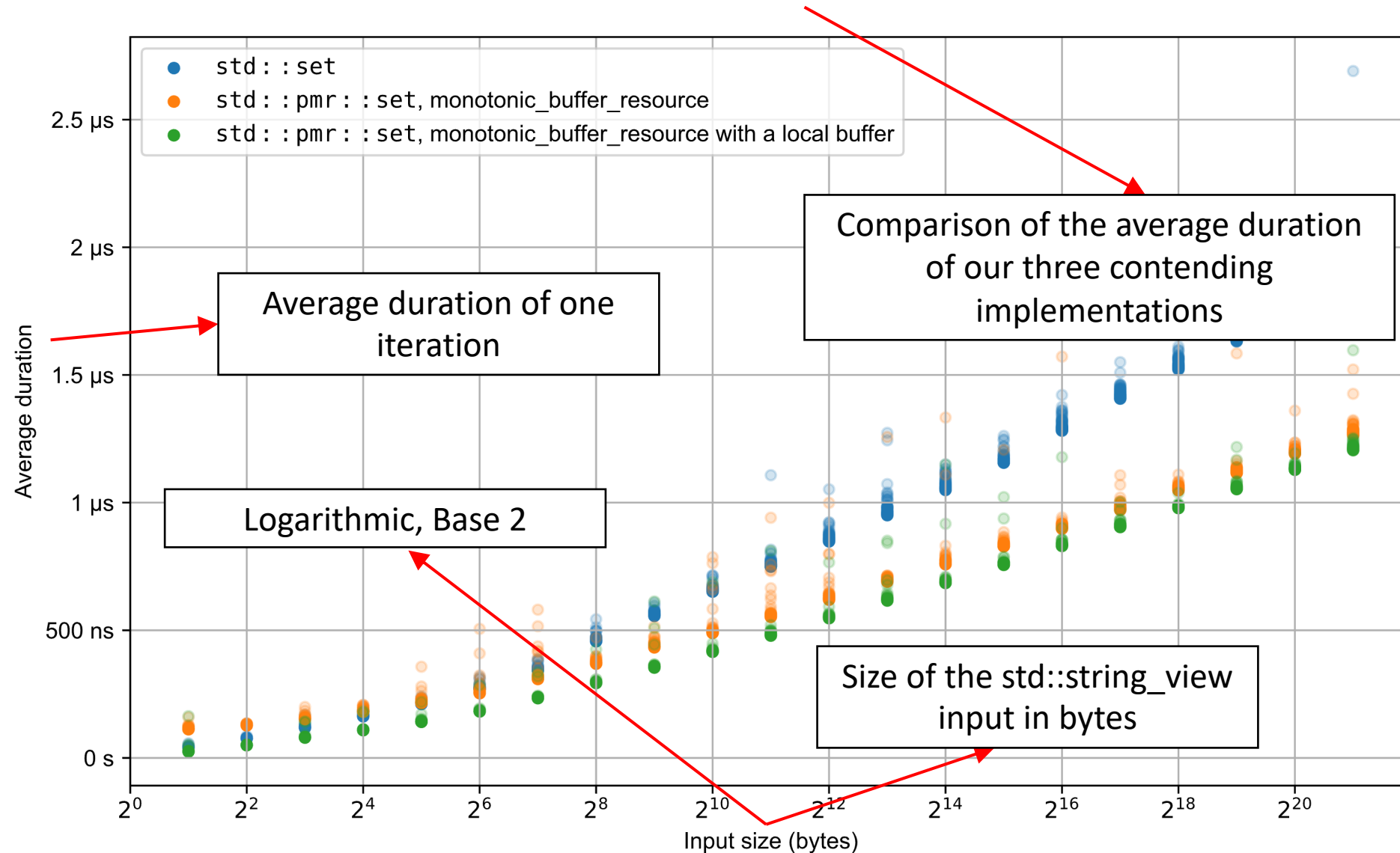
1.3 - Contender: Use stack for storage

```
std::size_t countUniqueChars3(const std::string_view& s)
{
    std::array<std::byte, 10'240> buffer;
    std::pmr::monotonic_buffer_resource mr(
        buffer.data(), buffer.size(), std::pmr::null_memory_resource());
    std::pmr::set<char> uniq(&mr);
    uniq.insert(s.begin(), s.end());
    return uniq.size();
}
```

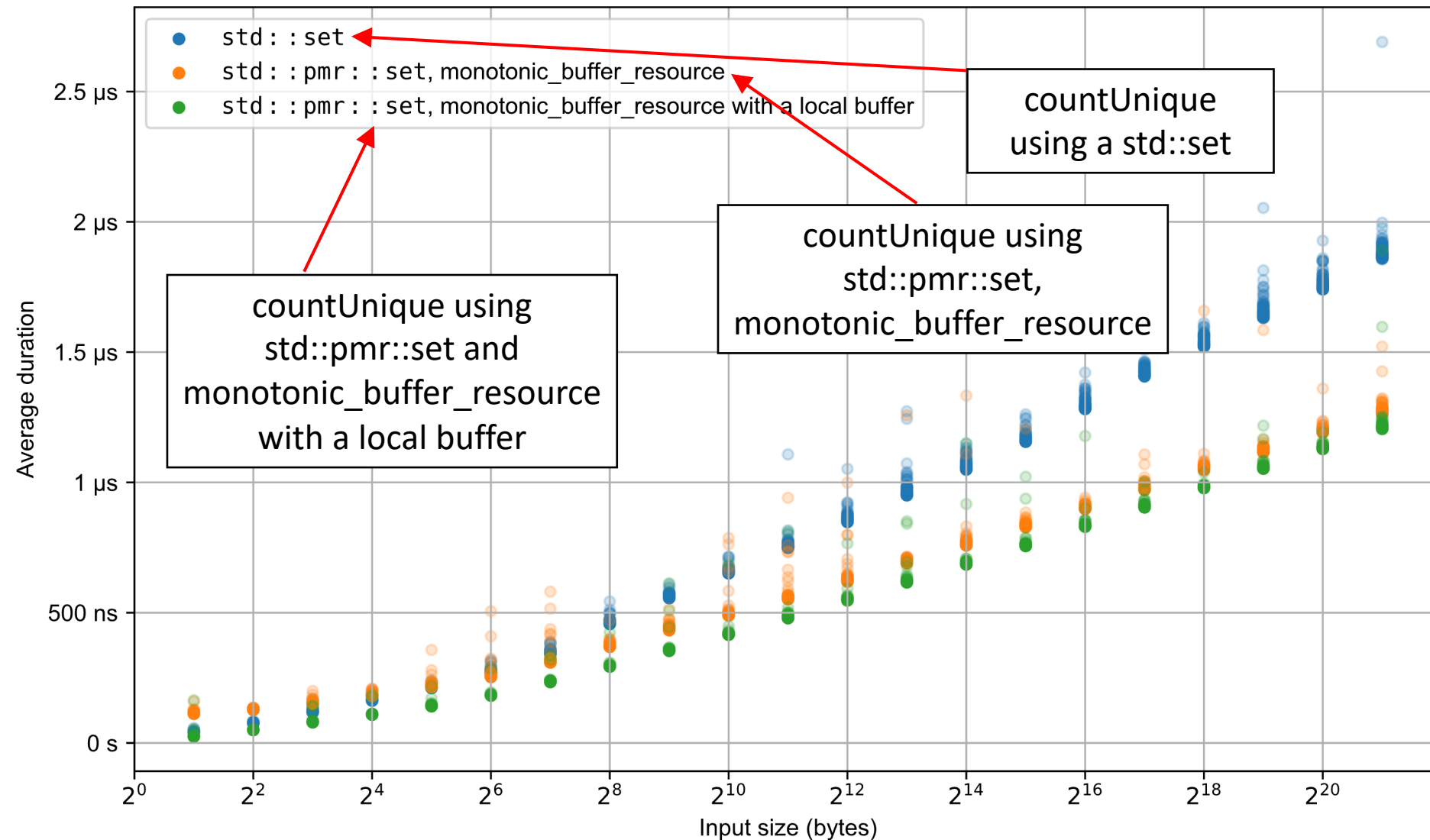

Allocator Impact on Runtime Performance

- ☒ Allocators and how they can improve performance?
- ☒ Allocator Performance Impact Analysis
- ☐ Results
- ☐ Conclusion

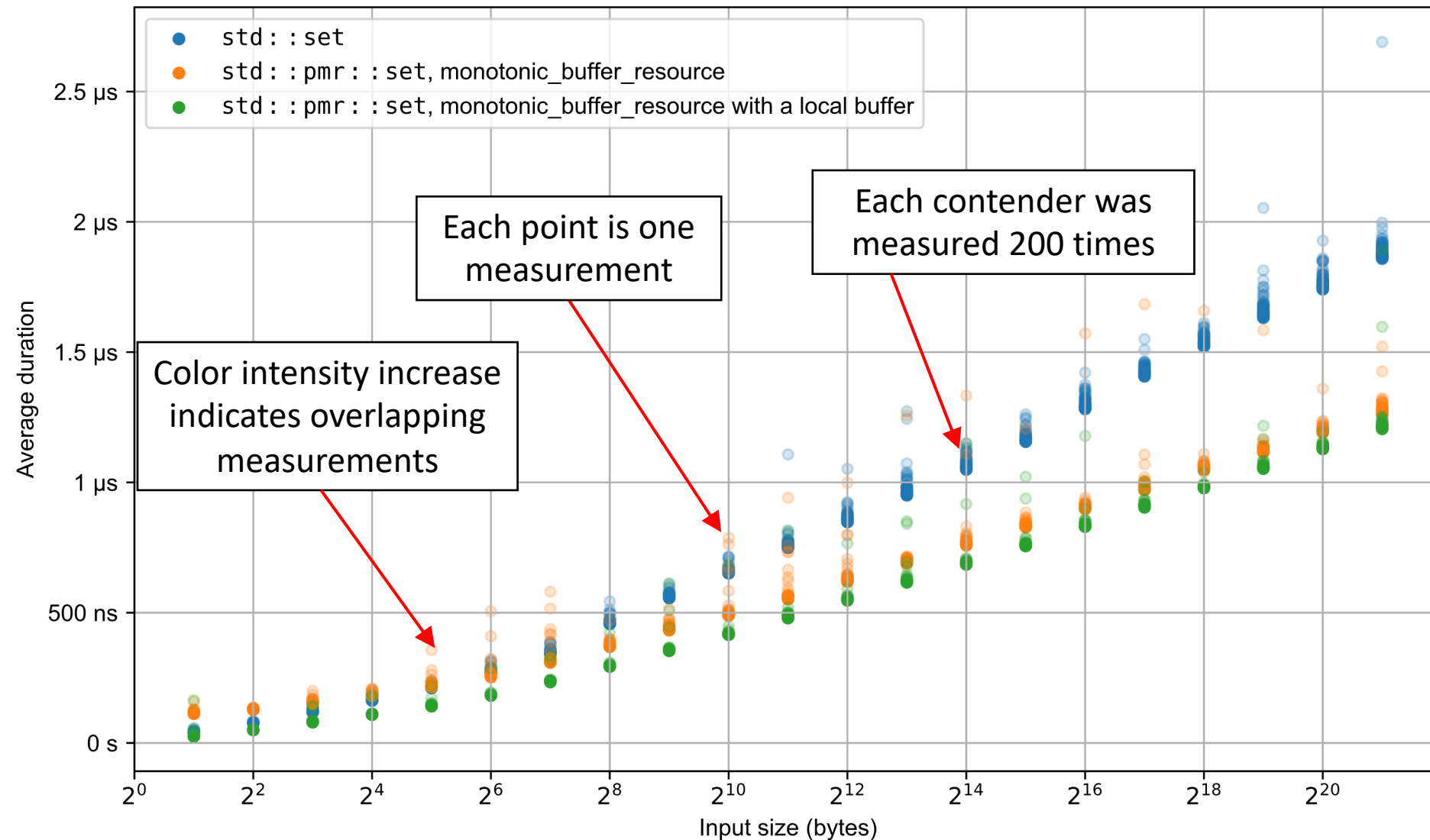
Comparison: countUnique run time



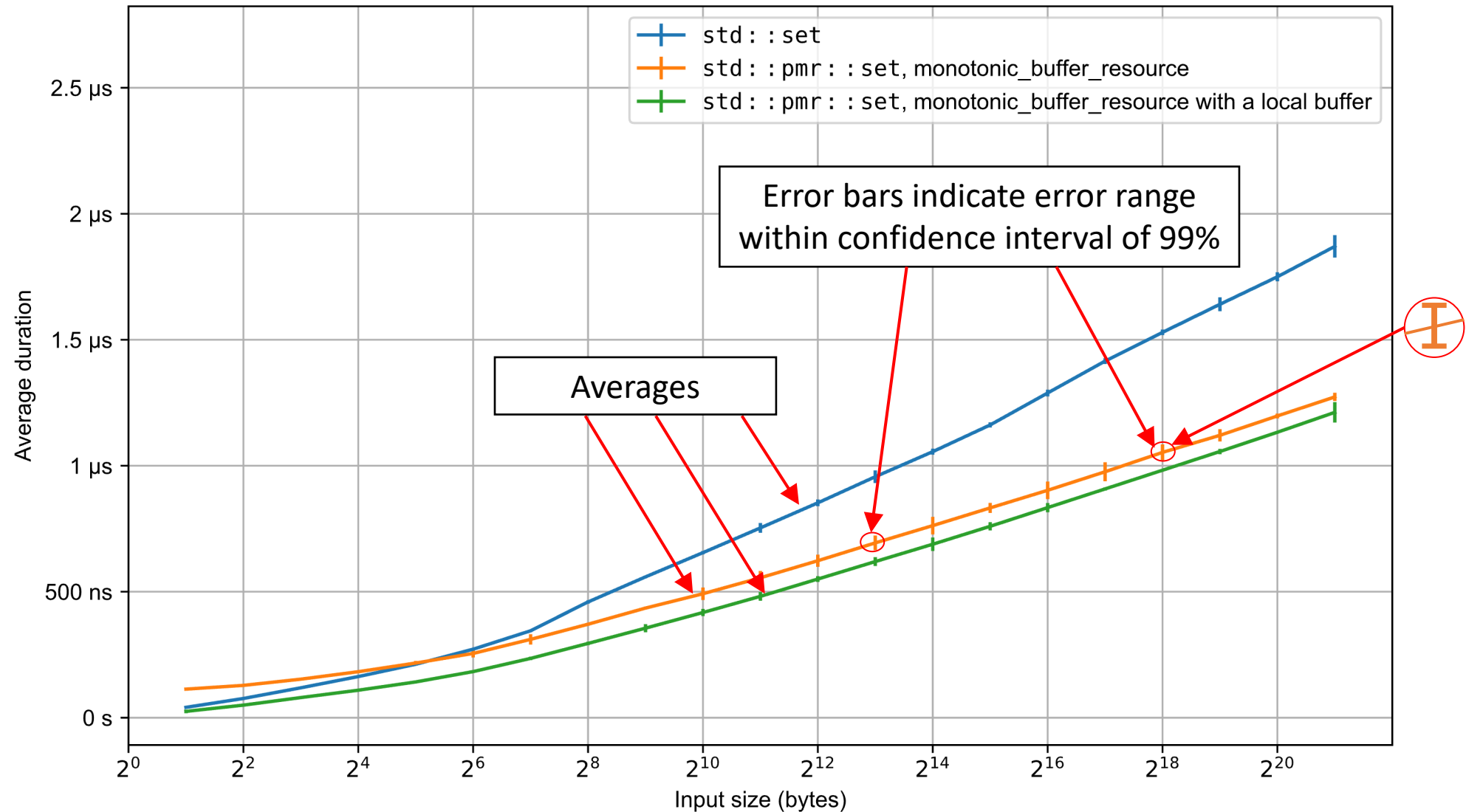
Comparison: countUnique run time



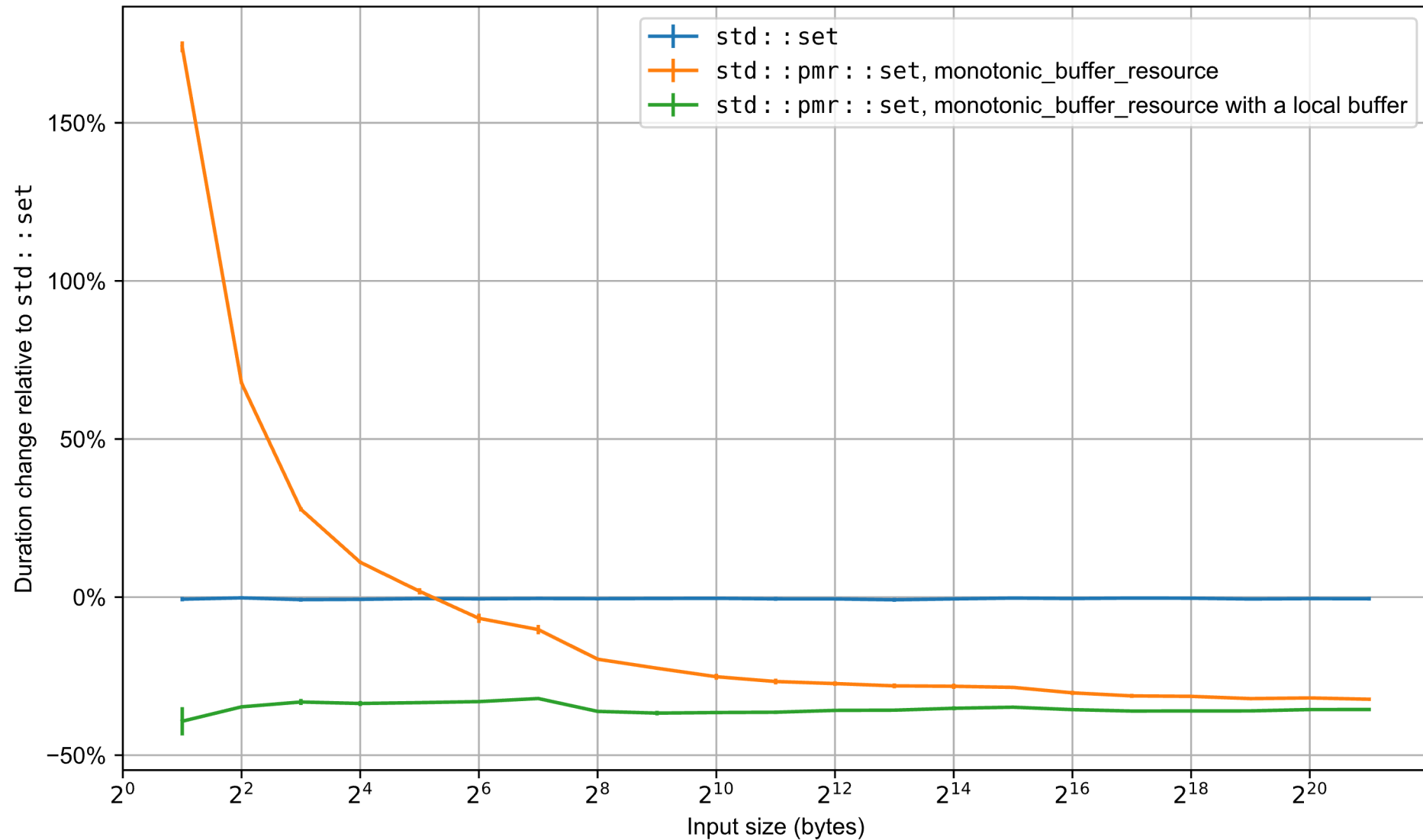
Comparison: countUnique run time



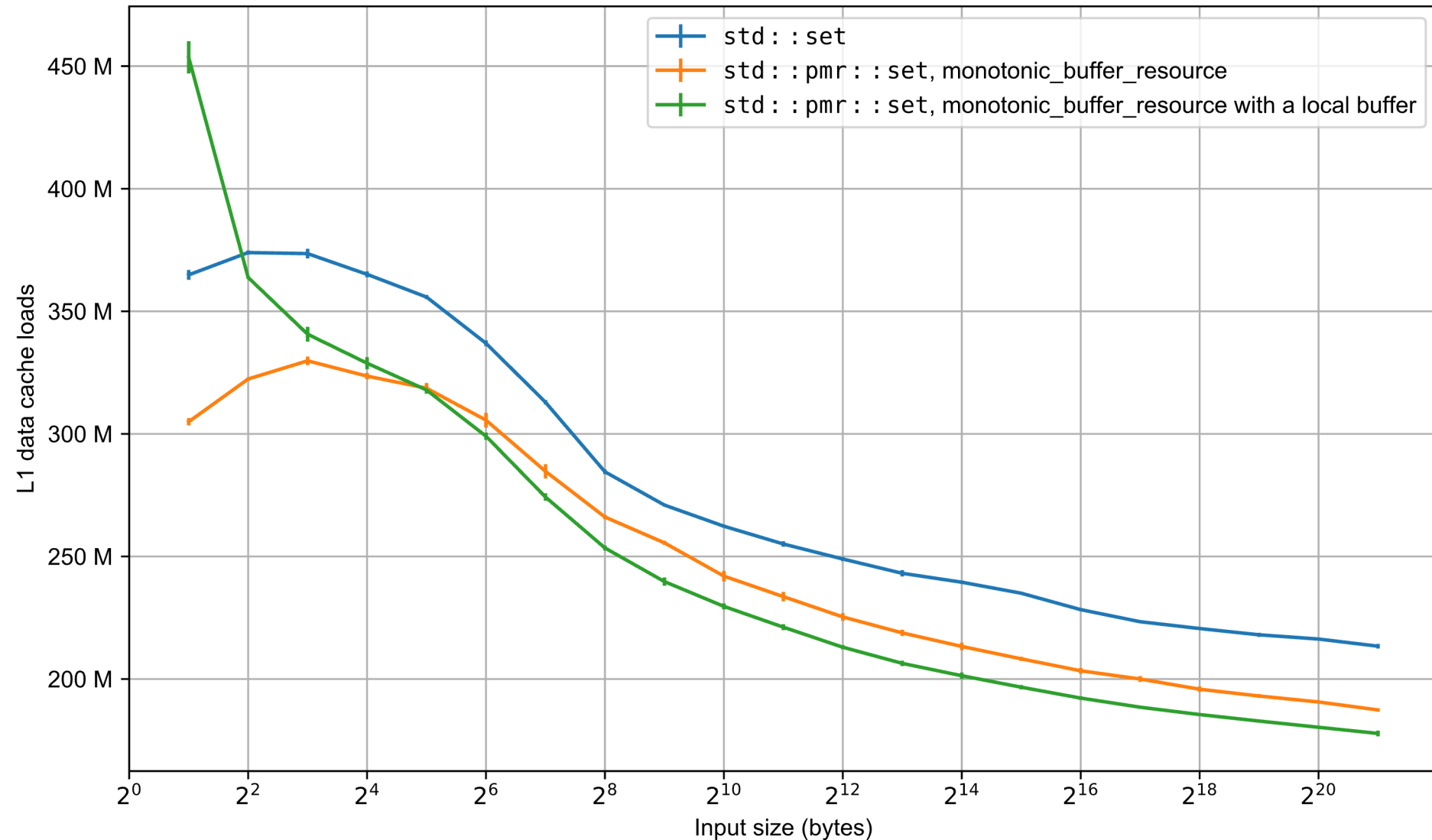
Comparison: countUnique run time



Comparison: Run time relative to `std::set`



Comparison: Memory loads



Case Study 1 - Performance

- Highlight the performance impact
 - Compare execution times
 - Compare contenders performance relative to the baseline
 - Effective visualization
- Performance metric

Simulate Allocation Diffusion: Heap littering

- Simulate the effect of a long-running program
 - We call it: “memory littering”
- Mission
 - Artificially reproduce the effect of allocation diffusion
 - Show impact of diffusion on our algorithms in long-running programs
- Options:
 - Controllable
 - When
 - How much

Heap littering algorithm

1. Instrument new and delete
 - Run our algorithm some number of times with delete disabled
 - Keep allocations in a static list
2. Randomly free a configurable fraction of the actual allocations
 - Emphasis: Random
3. Start benchmarking as normal
 - The Global heap is now fragmented.

How littering affects measurement

- We are not interested in littering performance itself.
 - Exclude the littering stage from performance measurements.
- Compare algorithm performance with a littered heap against the baseline (without littering).
- We used libpfm4.
- Reminder: Start and stop perf measurement on command.
 - Your program can communicate start and stop commands to perf via IPC.
 - Con: Perf wants the file descriptor(s).
 - VTune: `__itt_pause()` and `__itt_resume()` (in `ittnotify.h`)

Start and stop perf with a file descriptor

```
mkfifo ctl_fd.fifo ack_fd.fifo
```

```
exec {ctl_fd}<>ctl_fd.fifo
```

```
exec {ack_fd}<>ack_fd.fifo
```

```
perf stat --delay=-1 --control fd:${ctl_fd},${ack_fd} -- prog
```

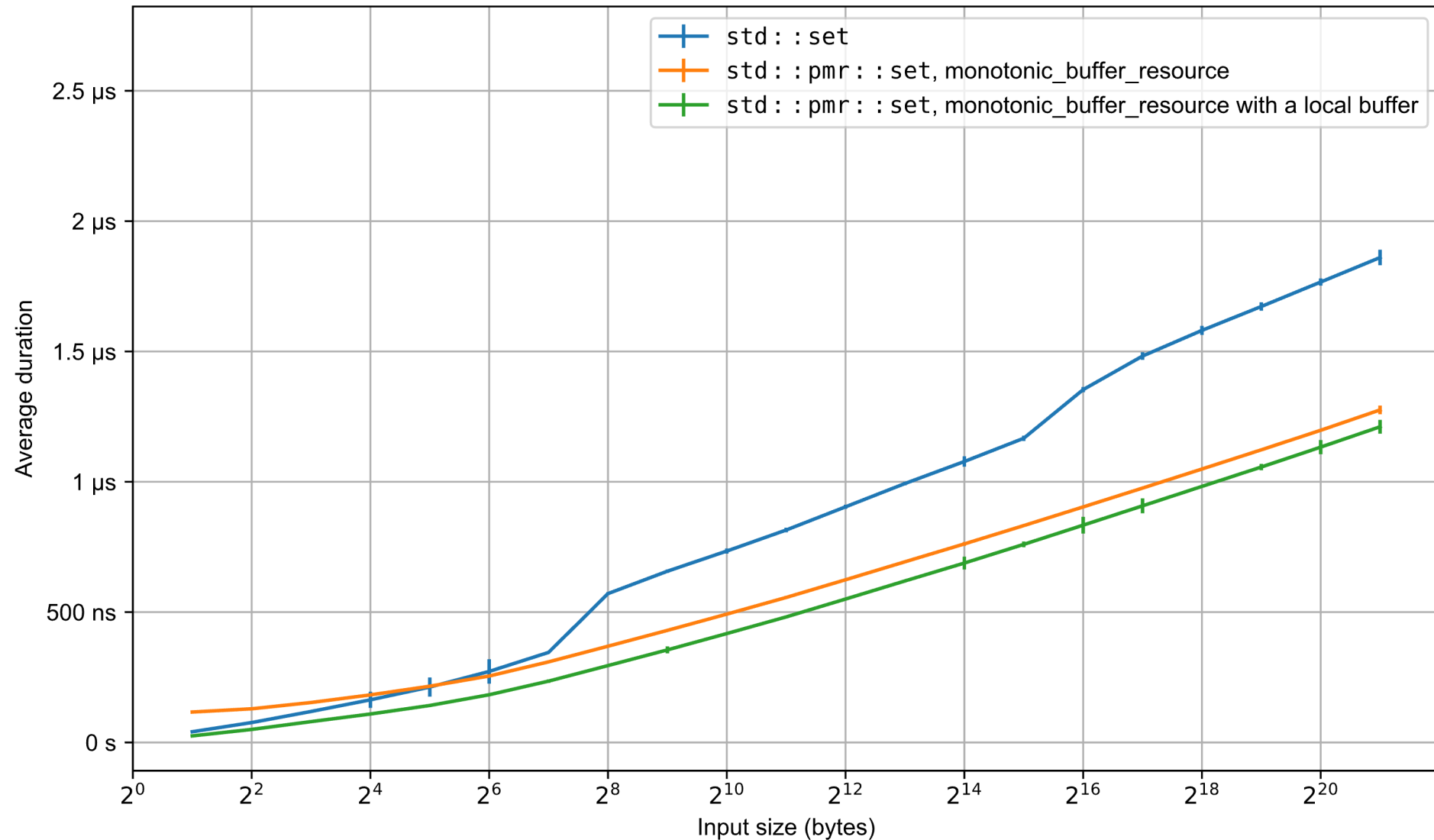
```
perf_pid=$!
```

```
sleep 4 # Wait 4 seconds
```

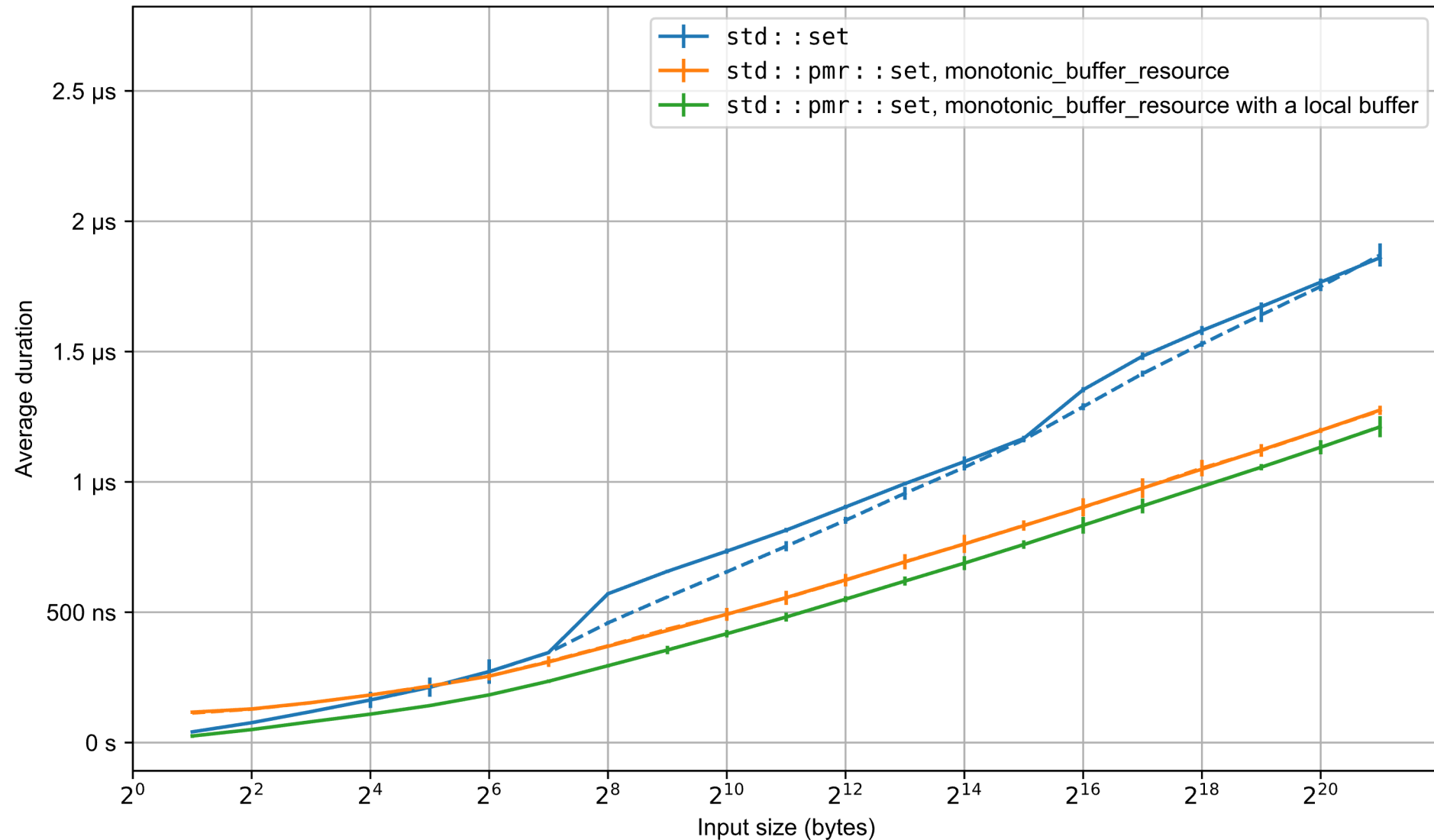
```
echo "enable" >&${ctl_fd} # Start measurements
```

```
read -u ${ack_fd} ack && echo "ack: $ack" # Read 'ack\n' from the ack_fd  
wait -n $perf_pid
```

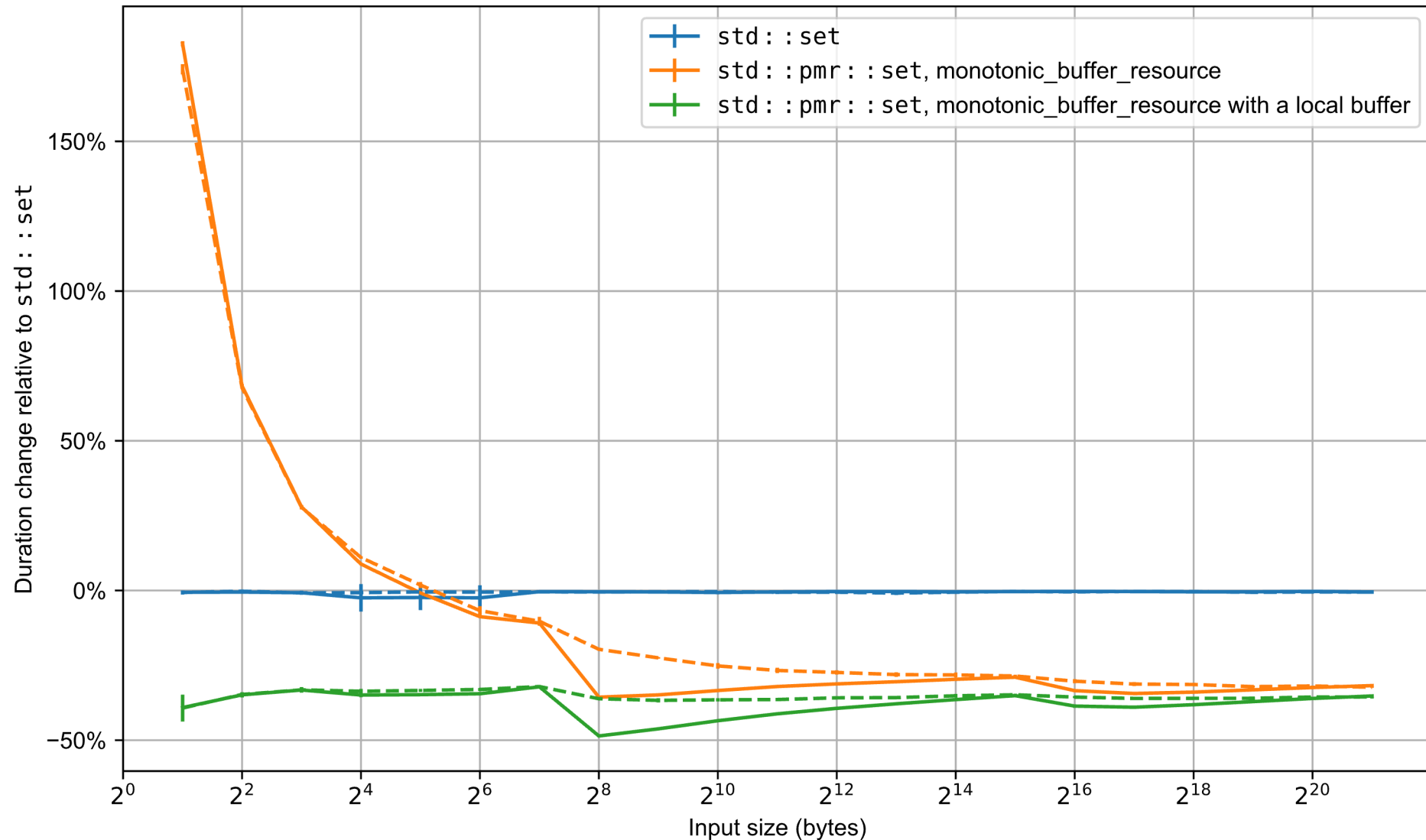
Comparison: countUnique run time, littered



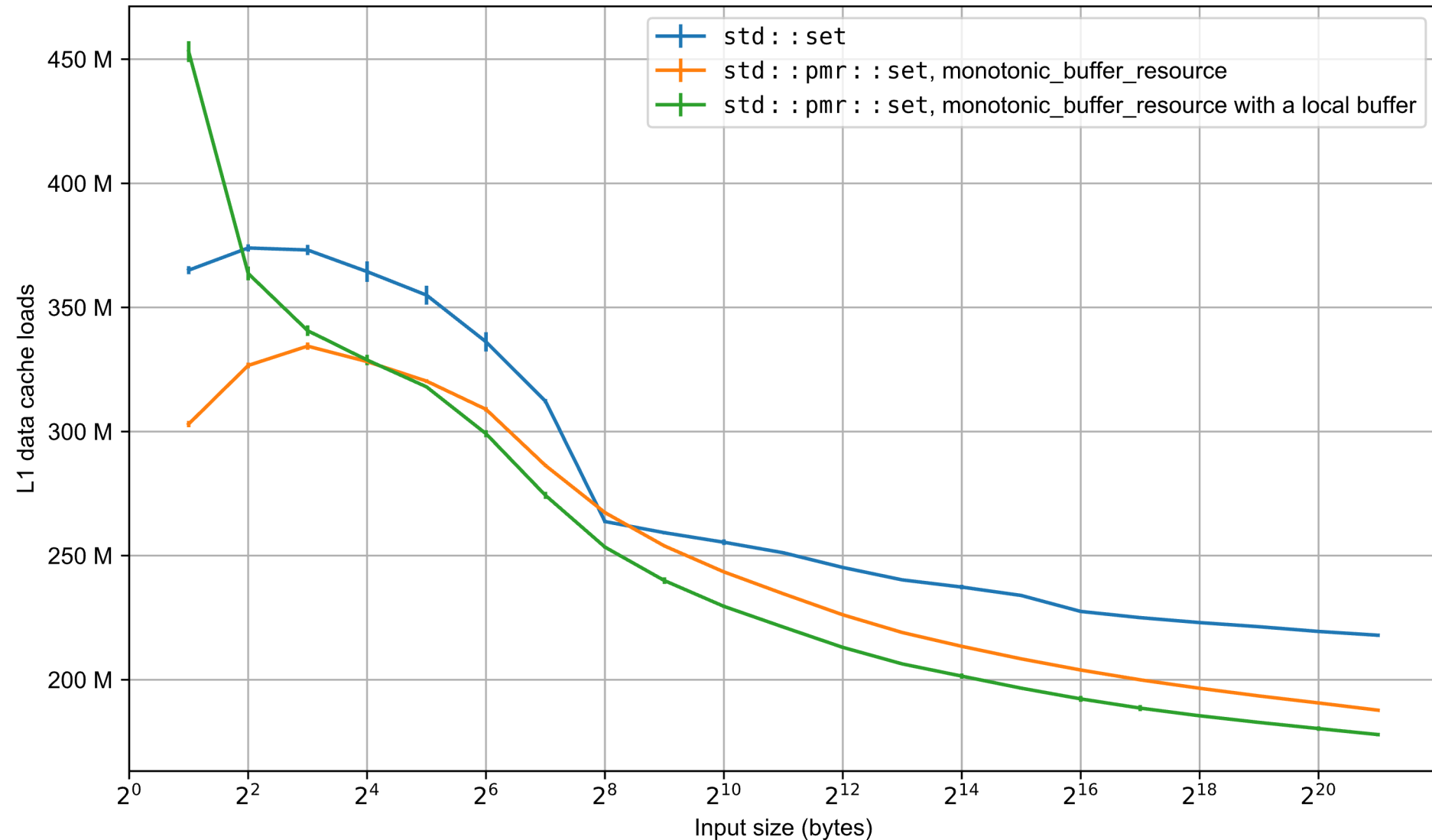
Comparison: Littered vs unlittered runtime



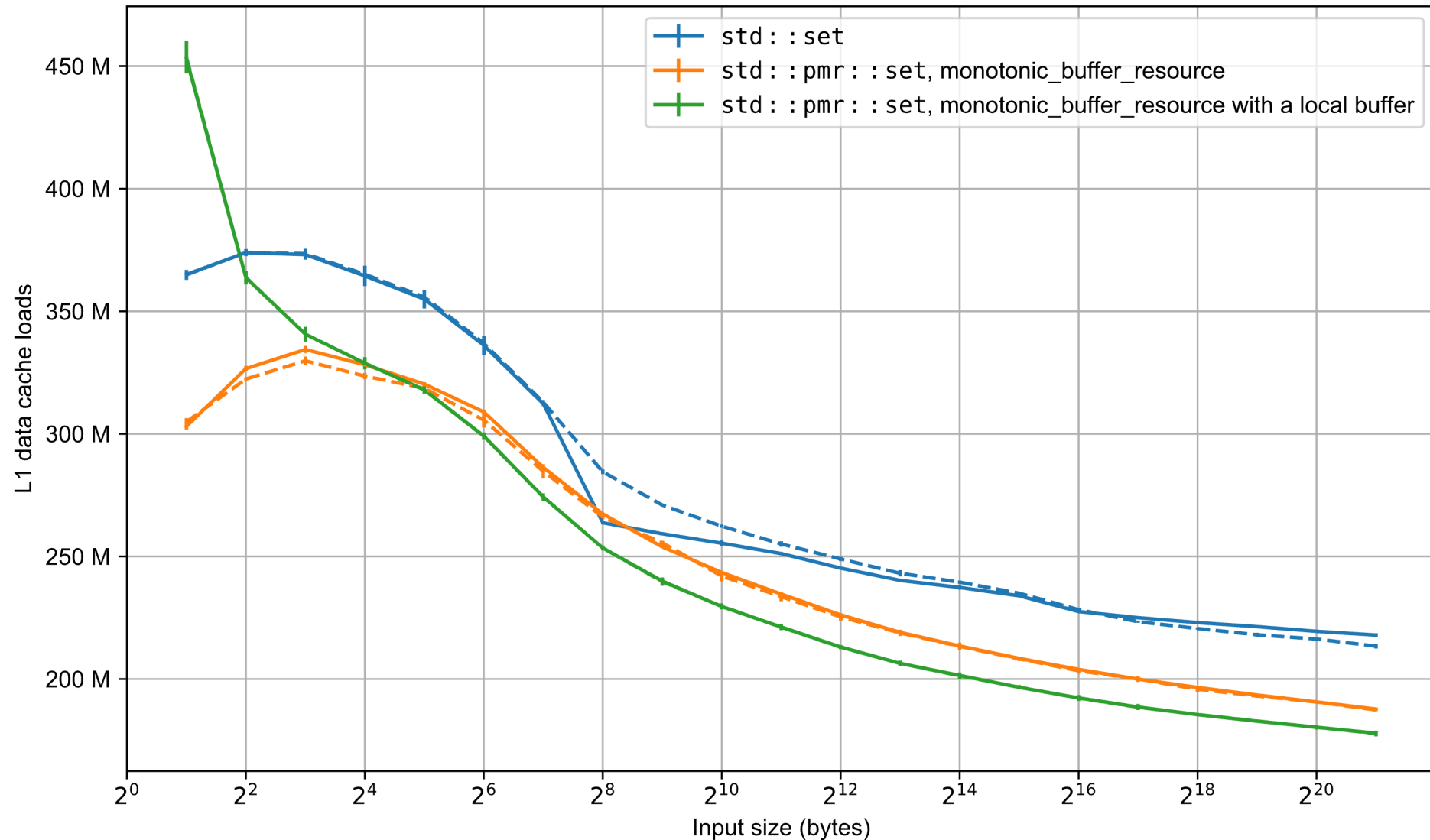
Comparison: Runtime relative to std::set



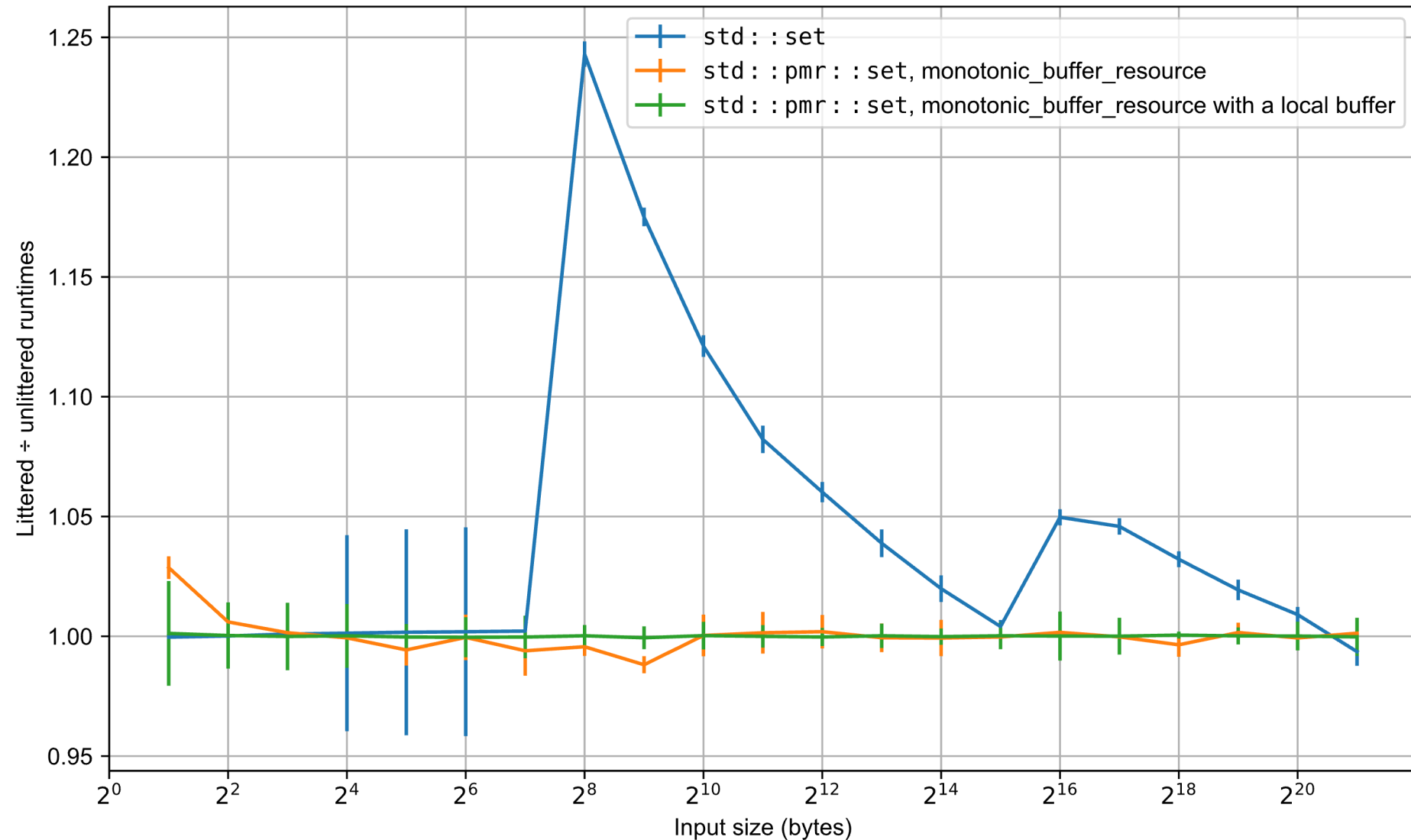
Comparison: Memory loads, littered



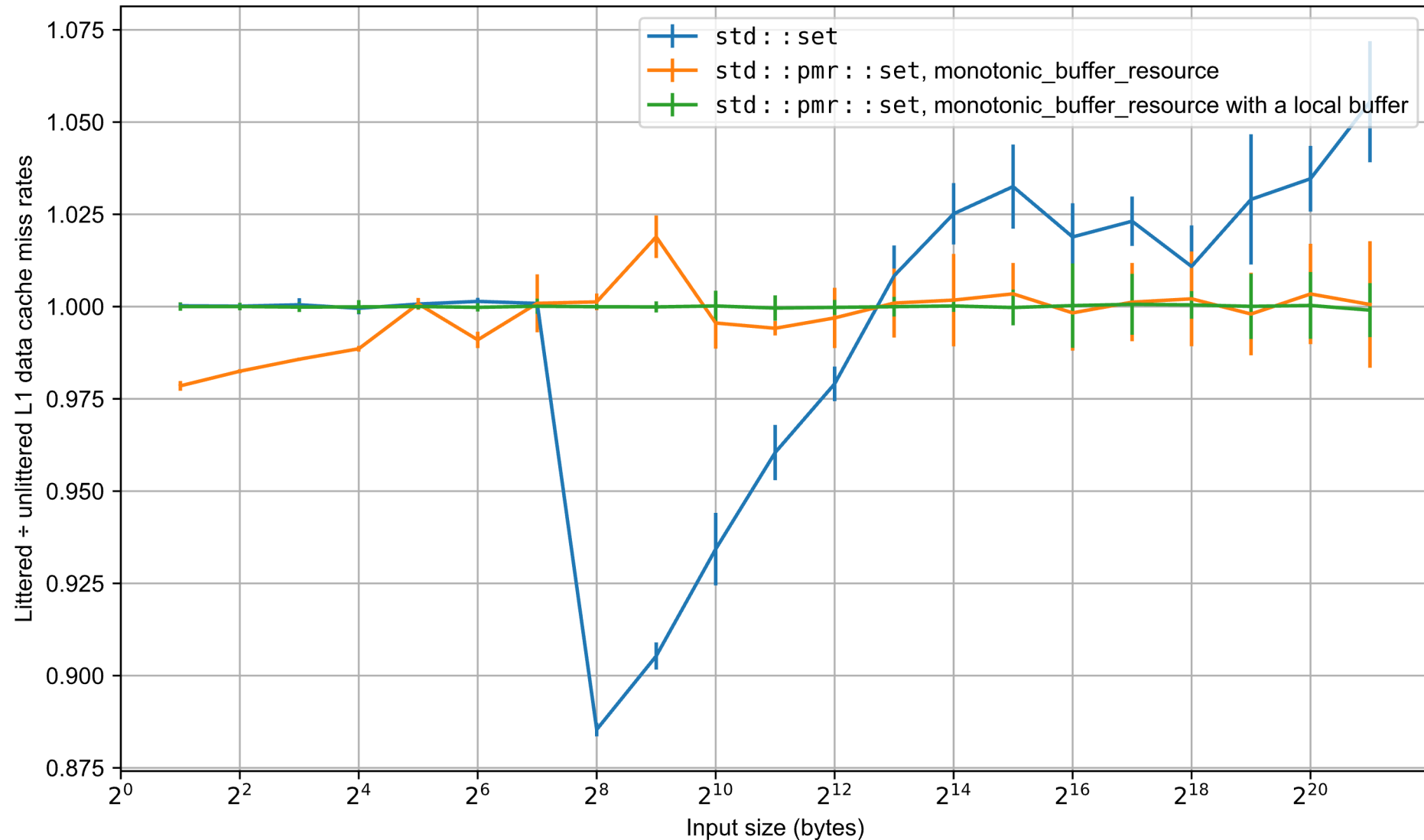
Comparison: Memory loads, littered



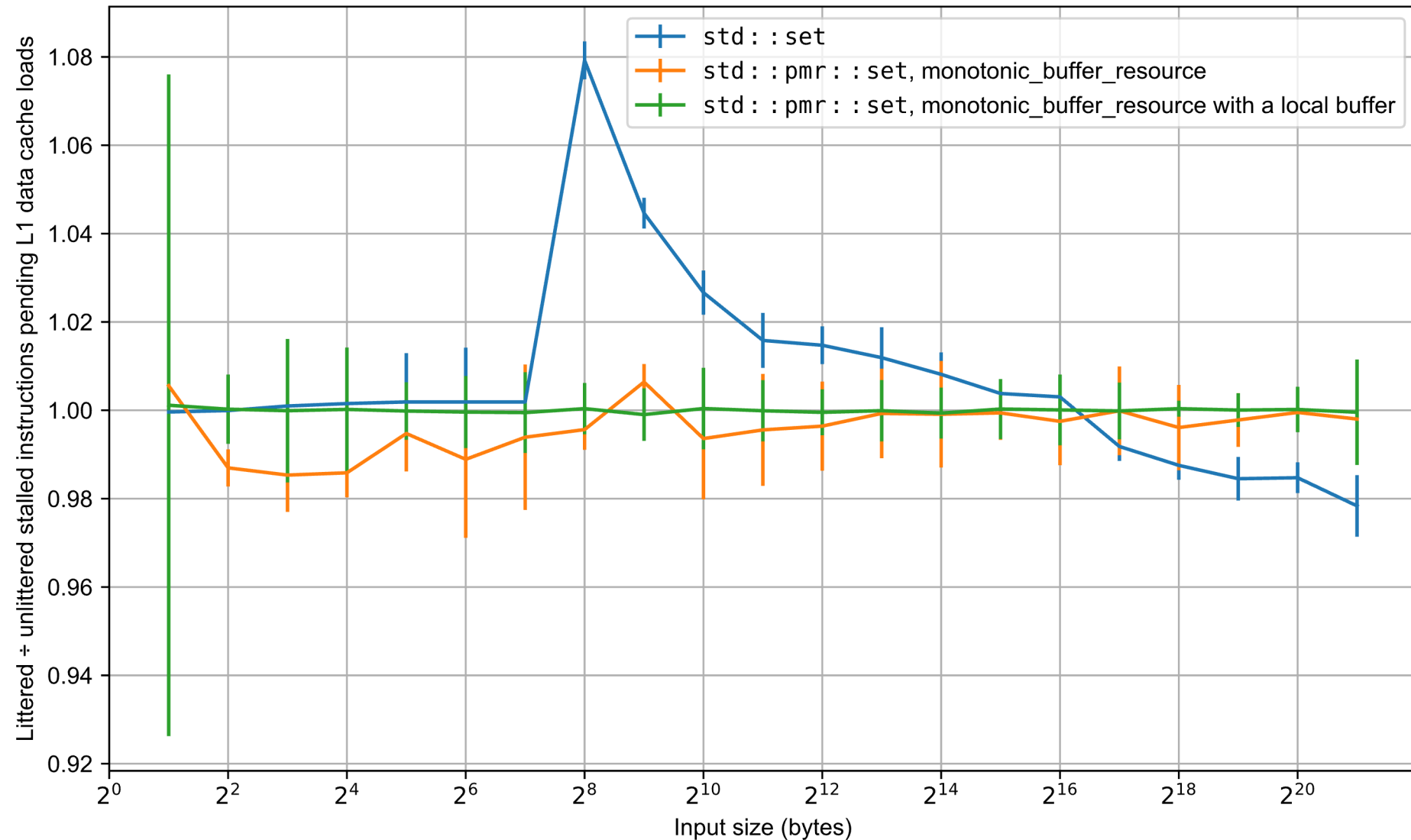
Comparison: Littered ÷ unlittered run time



Comparison: L1d miss rates, relative to baseline



Comparison: Littered ÷ unlittered stalled instructions pending L1d load



Case Study 2 - Objectives Review

- Highlight the performance impact
 - Compare execution times
 - Compare contenders performance relative to the baseline
 - Effective visualization
- Performance metric: Number of stalled instructions pending L1d load

Allocator Impact on Runtime Performance

- ☒ Allocators and how they can improve performance?
- ☒ Allocator Performance Impact Analysis
- ☒ Results
- ☐ Conclusion

Conclusion

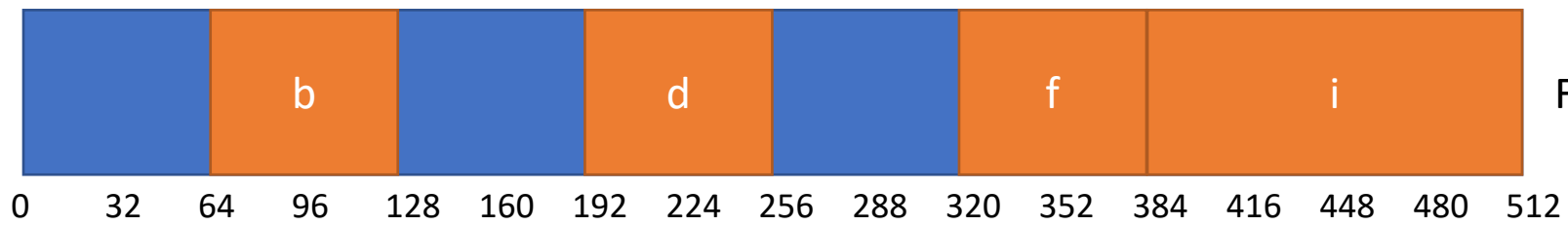
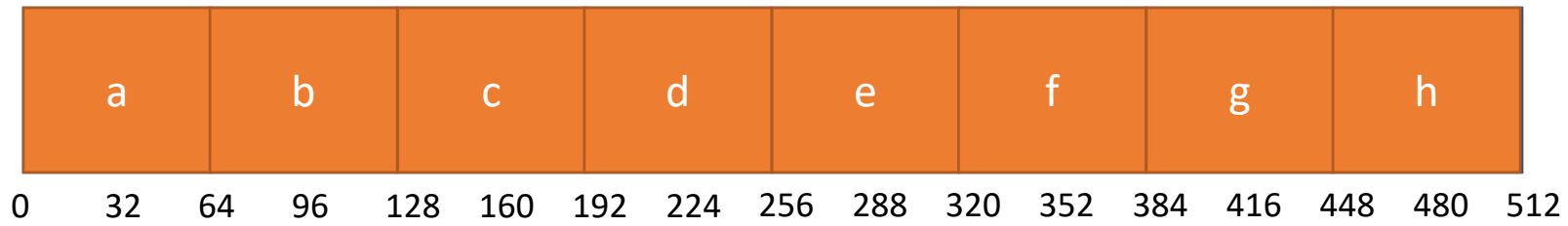
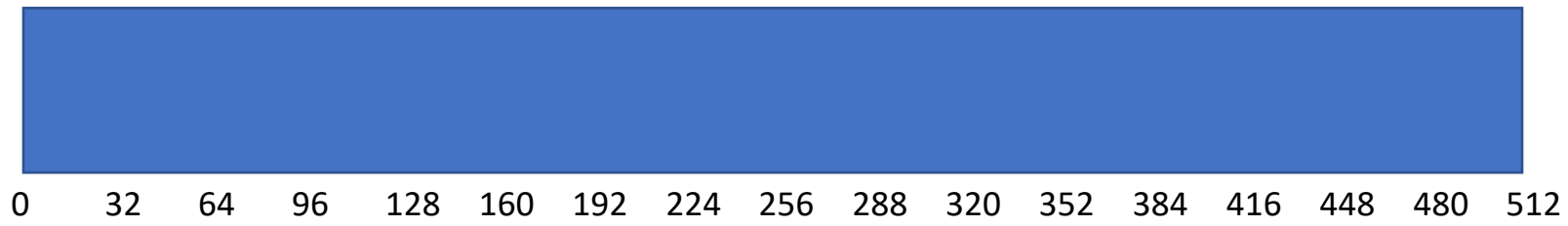
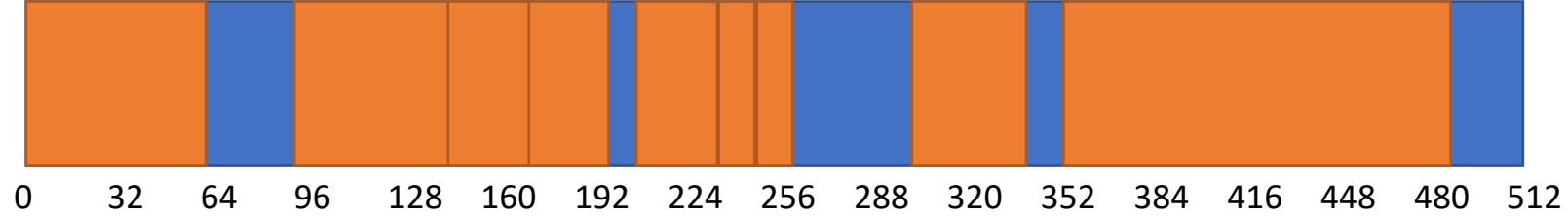
- Chose the correct tool
- Worked around the resolution and precision constraints of the instrumentation and profiling tools
- Designing effective presentation and visualization material.

Takeaway

- We showed a method for assessing the impact of allocators on performance.
- We simulated a long-running task by fragmenting a heap to produce allocation diffusion.
- Be excited about the book.
 - Authors: Joshua Berne, John Lakos
 - ISBN: 978-0-13-806072-5
 - Estimated release date: H2 2023
 - Look for an announcement in the fall.
 - Answer to John Lakos otherwise.

Questions?

Extras



Fragmented heap

■ Memory space ■ Allocated memory space

2.1 - Baseline: Use std::unordered_set

```
std::size_t countUniqueChars1U(const std::string_view& s)
{
```

```
    std::unordered_set<char> uniq;
    uniq.insert(s.begin(), s.end());
    return uniq.size();
}
```

2.2 - Contender: std::pmr::unordered_set

```
std::size_t countUniqueChars2U(const std::string_view& s)
{
    std::pmr::monotonic_buffer_resource mr;

    std::pmr::unordered_set<char> uniq(&mr);
    uniq.insert(s.begin(), s.end());
    return uniq.size();
}
```

Contender 4: Use std::bitset

```
std::size_t countUniqueChars4(const std::string_view& s)
{
    std::bitset<256> bits;
    for (const char c : s) {
        bits.set(c & 0xff);
    }
    return bits.count();
}
```

Contender 4A: Use std::array

```
std::size_t countUniqueChars4A(const std::string_view& s)
{
    std::array<std::byte, 256> bits{};
    for (const char c : s) {
        bits[c & 0xff] = static_cast<std::byte>(1);
    }
    return std::count(bits.begin(), bits.end(),
                     static_cast<std::byte>(1));
}
```

Survey – Memory Use Statistics

- Working set sizes
- Page faults
- LLC accesses and misses
- L1d accesses and misses